koa

中文文档

目錄

koa中文文档	0
简介	1
安装	2
应用	3
Context(上下文)	4
请求(Request)	5
响应(Response)	6
相关资源	7

koa 中文文档

> 来源：koa 中文文档

简介

koa 是由 Express 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 Web 框架。使用 koa 编写 web 应用，通过组合不同的 generator，可以免除重复繁琐的回调函数嵌套，并极大地提升错误处理的效率。koa 不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写 Web 应用变得得心应手。

安装

Koa 目前需要 >=0.11.x版本的 node 环境。并需要在执行 node 的时候附带 --harmony 来引入 generators 。 如果您安装了较旧版本的 node ，您可以安装 n (node版本控制器)，来快速安装 0.11.x

```
$ npm install -g n
$ n 0.11.12
$ node --harmony my-koa-app.js
```

应用

Koa 应用是一个包含一系列中间件 generator 函数的对象。 这些中间件函数基于 request 请求以一个类似于栈的结构组成并依次执行。 Koa 类似于其他中间件系统（比如 Ruby's Rack 、Connect 等）， 然而 Koa 的核心设计思路是为中间件层提供高级语法糖封装，以增强其互用性和健壮性，并使得编写中间件变得相当有趣。

Koa 包含了像 content-negotiation（内容协商）、cache freshness（缓存刷新）、proxy support（代理支持）和 redirection（重定向）等常用任务方法。 与提供庞大的函数支持不同，Koa只包含很小的一部分，因为Koa并不绑定任何中间件。

任何教程都是从 hello world 开始的，Koa也不例外^_^

```
var koa = require('koa');
var app = koa();

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

中间件级联

Koa 的中间件通过一种更加传统（您也许会很熟悉）的方式进行级联，摒弃了以往 node 频繁的回调函数造成的复杂代码逻辑。 我们通过 generators 来实现“眞正”的中间件。 Connect 简单地将控制权交给一系列函数来处理，直到函数返回。 与之不同，当执行到 `yield next` 语句时，Koa 暂停了该中间件，继续执行下一个符合请求的中间件('downstrem')，然后控制权再逐级返回给上层中间件('upstream')。

下面的例子在页面中返回 "Hello World"，然而当请求开始时，请求先经过 `x-response-time` 和 `logging` 中间件，并记录中间件执行起始时间。 然后将控制权交给 `reponse` 中间件。当中间件运行到 `yield next` 时，函数挂起并将控制前交给下一个中间件。当没有中间件执行 `yield next` 时，程序栈会逆序唤起被挂起的中间件来执行接下来的代码。

```
var koa = require('koa');
var app = koa();

// x-response-time

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  this.set('X-Response-Time', ms + 'ms');
});

// logger

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

// response

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

配置

应用配置是 `app` 实例属性，目前支持的配置项如下：

- `app.name` 应用名称（可选项）
- `app.env` 默认为 **NODE_ENV** 或者 `development`
- `app.proxy` 如果为 `true`，则解析 "Host" 的 header 域，并支持 `X-Forwarded-Host`
- `app.subdomainOffset` 默认为2，表示 `.subdomains` 所忽略的字符偏移量。

app.listen(...)

Koa 应用并非是一个 1-to-1 表征关系的 HTTP 服务器。 一个或多个Koa应用可以被挂载到一起组成一个包含单一 HTTP 服务器的大型应用群。

如下为一个绑定3000端口的简单 Koa 应用，其创建并返回了一个 HTTP 服务器，为 `Server#listen()` 传递指定参数（参数的详细文档请查看nodejs.org）。

```
var koa = require('koa');
var app = koa();
app.listen(3000);
```

`app.listen(...)` 实际上是以下代码的语法糖:

```
var http = require('http');
var koa = require('koa');
var app = koa();
http.createServer(app.callback()).listen(3000);
```

这意味着您可以同时支持 HTTPS 和 HTTPS，或者在多个端口监听同一个应用。

```
var http = require('http');
var koa = require('koa');
var app = koa();
http.createServer(app.callback()).listen(3000);
http.createServer(app.callback()).listen(3001);
```

app.callback()

返回一个适合 `http.createServer()` 方法的回调函数用来处理请求。 您也可以使用这个回调函数将您的app挂载在 Connect/Express 应用上。

app.use(function)

为应用添加指定的中间件，详情请看 Middleware

app.keys=

设置签名Cookie密钥，该密钥会被传递给 KeyGrip。

当然，您也可以自己生成 `KeyGrip` 实例：

```
app.keys = ['im a newer secret', 'i like turtle'];
app.keys = new KeyGrip(['im a newer secret', 'i like turtle'], 'sha256');
```

在进行cookie签名时，只有设置 `signed` 为 `true` 的时候，才会使用密钥进行加密：

```
this.cookies.set('name', 'tobi', { signed: true });
```

错误处理

默认情况下Koa会将所有错误信息输出到 stderr，除非 **NODE_ENV** 是 "test"。为了实现自定义错误处理逻辑（比如 centralized logging），您可以添加 "error" 事件监听器。

```
app.on('error', function(err){
  log.error('server error', err);
});
```

如果错误发生在 请求/响应 环节，并且其不能够响应客户端时，`Contenxt` 实例也会被传递到 `error` 事件监听器的回调函数里。

```
app.on('error', function(err, ctx){
  log.error('server error', err, ctx);
});
```

当发生错误但仍能够响应客户端时（比如没有数据写到socket中），Koa会返回一个500错误(Internal Server Error)。

无论哪种情况，Koa都会生成一个应用级别的错误信息，以便实现日志记录等目的。

Context(上下文)

Koa Context 将 node 的 `request` 和 `response` 对象封装在一个单独的对象里面，其为编写 web 应用和 API 提供了很多有用的方法。

这些操作在 HTTP 服务器开发中经常使用，因此其被添加在上下文这一层，而不是更高层框架中，因此将迫使中间件需要重新实现这些常用方法。

`context` 在每个 request 请求中被创建，在中间件中作为接收器(receiver)来引用，或者通过 `this` 标识符来引用：

```
app.use(function *(){
  this; // is the Context
  this.request; // is a koa Request
  this.response; // is a koa Response
});
```

许多 context 的访问器和方法为了便于访问和调用，简单的委托给他们的 `ctx.request` 和 `ctx.response` 所对应的等价方法， 比如说 `ctx.type` 和 `ctx.length` 代理了 `response` 对象中对应的方法，`ctx.path` 和 `ctx.method` 代理了 `request` 对象中对应的方法。

API

`Context` 详细的方法和访问器。

ctx.req

Node 的 `request` 对象。

ctx.res

Node 的 `response` 对象。

Koa 不支持 直接调用底层 res 进行响应处理。请避免使用以下 node 属性：

- `res.statusCode`
- `res.writeHead()`
- `res.write()`
- `res.end()`

ctx.request

Koa 的 `Request` 对象。

ctx.response

Koa 的 `Response` 对象。

ctx.app

应用实例引用。

ctx.cookies.get(name, [options])

获得 cookie 中名为 `name` 的值，`options` 为可选参数：

- 'signed': 如果为 true，表示请求时 cookie 需要进行签名。

注意：Koa 使用了 Express 的 cookies 模块，options 参数只是简单地直接进行传递。

ctx.cookies.set(name, value, [options])

设置 cookie 中名为 `name` 的值，`options` 为可选参数：

- `signed`: 是否要做签名
- `expires`: cookie 有效期时间
- `path`: cookie 的路径，默认为 `/'`
- `domain`: cookie 的域
- `secure`: false 表示 cookie 通过 HTTP 协议发送，true 表示 cookie 通过 HTTPS 发送。
- `httpOnly`: true 表示 cookie 只能通过 HTTP 协议发送

注意：Koa 使用了 Express 的 cookies 模块，options 参数只是简单地直接进行传递。

ctx.throw(msg, [status])

抛出包含 `.status` 属性的错误，默认为 `500`。该方法可以让 Koa 准确的响应处理状态。Koa支持以下组合：

```
this.throw(403)
this.throw('name required', 400)
this.throw(400, 'name required')
this.throw('something exploded')
```

`this.throw('name required', 400)` 等价于：

```
var err = new Error('name required');
err.status = 400;
throw err;
```

注意：这些用户级错误被标记为 `err.expose` ，其意味着这些消息被准确描述为对客户端的响应，而并非使用在您不想泄露失败细节的场景中。

ctx.respond

为了避免使用 Koa 的内置响应处理功能，您可以直接赋值 `this.repond = false;` 。如果您不想让 Koa 来帮助您处理 reponse，而是直接操作原生 `res` 对象，那么请使用这种方法。

注意： 这种方式是不被 Koa 支持的。其可能会破坏 Koa 中间件和 Koa 本身的一些功能。其只作为一种 hack 的方式，并只对那些想要在 Koa 方法和中间件中使用传统 `fn(req, res)` 方法的人来说会带来便利。

Request aliases

以下访问器和别名与 Request 等价：

- `ctx.header`
- `ctx.method`
- `ctx.method=`
- `ctx.url`
- `ctx.url=`
- `ctx.originalUrl`
- `ctx.path`
- `ctx.path=`
- `ctx.query`
- `ctx.query=`
- `ctx.querystring`
- `ctx.querystring=`
- `ctx.host`
- `ctx.hostname`
- `ctx.fresh`
- `ctx.stale`
- `ctx.socket`
- `ctx.protocol`
- `ctx.secure`
- `ctx.ip`
- `ctx.ips`

- `ctx.subdomains`
- `ctx.is()`
- `ctx.accepts()`
- `ctx.acceptsEncodings()`
- `ctx.acceptsCharsets()`
- `ctx.acceptsLanguages()`
- `ctx.get()`

Response aliases

以下访问器和别名与 Response 等价：

- `ctx.body`
- `ctx.body=`
- `ctx.status`
- `ctx.status=`
- `ctx.length=`
- `ctx.length`
- `ctx.type=`
- `ctx.type`
- `ctx.headerSent`
- `ctx.redirect()`
- `ctx.attachment()`
- `ctx.set()`
- `ctx.remove()`
- `ctx.lastModified=`
- `ctx.etag=`

请求(Request)

Koa `Request` 对象是对 node 的 request 进一步抽象和封装，提供了日常 HTTP 服务器开发中一些有用的功能。

API

req.header

请求头对象

req.method

请求方法

req.method=

设置请求方法，在实现中间件时非常有用，比如 `methodOverride()` 。

req.length

以数字的形式返回 request 的内容长度(Content-Length)，或者返回 `undefined` 。

req.url

获得请求url地址。

req.url=

设置请求地址，用于重写url地址时。

req.originalUrl

获取请求原始地址。

req.path

获取请求路径名。

req.path=

设置请求路径名，并保留请求参数(就是url中?后面的部分)。

req.querystring

获取查询参数字符串(url中?后面的部分)，不包含 ?。

req.querystring=

设置查询参数。

req.search

获取查询参数字符串，包含 ?。

req.search=

设置查询参数字符串。

req.host

获取 host (hostname:port)。 当 `app.proxy` 设置为 **true** 时，支持 `X-Forwarded-Host` 。

req.hostname

获取 hostname。当 `app.proxy` 设置为 **true** 时，支持 `X-Forwarded-Host` 。

req.type

获取请求 `Content-Type` ，不包含像 "charset" 这样的参数。

```
var ct = this.request.type;
// => "image/png"
```

req.charset

获取请求 charset，没有则返回 `undefined`:

```
this.request.charset
// => "utf-8"
```

req.query

将查询参数字符串进行解析并以对象的形式返回，如果没有查询参数字字符串则返回一个空对象。

注意：该方法不支持嵌套解析。

比如 "color=blue&size=small":

```
{
  color: 'blue',
  size: 'small'
}
```

req.query=

根据给定的对象设置查询参数字符串。

注意：该方法不支持嵌套对象。

```
this.query = { next: '/login' };
```

req.fresh

检查请求缓存是否 "fresh"(内容没有发生变化)。该方法用于在 `If-None-Match` / `ETag`, `If-Modified-Since` 和 `Last-Modified` 中进行缓存协调。当在 response headers 中设置一个或多个上述参数后，该方法应该被使用。

```
this.set('ETag', '123');

// cache is ok
if (this.fresh) {
  this.status = 304;
  return;
}

// cache is stale
// fetch new data
this.body = yield db.find('something');
```

req.stale

与 `req.fresh` 相反。

req.protocol

返回请求协议，"https" 或者 "http"。 当 `app.proxy` 设置为 **true** 时，支持 `X-Forwarded-Host` 。

req.secure

简化版 `this.protocol == "https"` ，用来检查请求是否通过 TLS 发送。

req.ip

请求远程地址。 当 `app.proxy` 设置为 **true** 时，支持 `X-Forwarded-Host` 。

req.ips

当 `X-Forwarded-For` 存在并且 `app.proxy` 有效，将会返回一个有序（从 upstream 到 downstream）ip 数组。 否则返回一个空数组。

req.subdomains

以数组形式返回子域名。

子域名是在host中逗号分隔的主域名前面的部分。默认情况下，应用的域名假设为host中最后两部分。其可通过设置 `app.subdomainOffset` 进行更改。

举例来说，如果域名是 "tobi.ferrets.example.com":

如果没有设置 `app.subdomainOffset` ，其 subdomains 为 `["ferrets", "tobi"]` 。 如果设置 `app.subdomainOffset` 为3，其 subdomains 为 `["tobi"]` 。

req.is(type)

检查请求所包含的 "Content-Type" 是否为给定的 type 值。 如果没有 request body，返回 `undefined` 。 如果没有 content type，或者匹配失败，返回 `false` 。 否则返回匹配的 content-type。

```
// With Content-Type: text/html; charset=utf-8
this.is('html'); // => 'html'
this.is('text/html'); // => 'text/html'
this.is('text/*', 'text/html'); // => 'text/html'

// When Content-Type is application/json
this.is('json', 'urlencoded'); // => 'json'
this.is('application/json'); // => 'application/json'
this.is('html', 'application/*'); // => 'application/json'

this.is('html'); // => false
```

比如说您希望保证只有图片发送给指定路由：

```
if (this.is('image/*')) {
  // process
} else {
  this.throw(415, 'images only!');
}
```

Content Negotiation

Koa `request` 对象包含 content negotiation 功能（由 accepts 和 negotiator 提供）：

- `req.accepts(types)`
- `req.acceptsEncodings(types)`
- `req.acceptsCharsets(charsets)`
- `req.acceptsLanguages(langs)`

如果没有提供 types，将会返回所有的可接受类型。

如果提供多种 types，将会返回最佳匹配类型。如果没有匹配上，则返回 `false`，您应该给客户端返回 `406 "Not Acceptable"`。

为了防止缺少 accept headers 而导致可以接受任意类型，将会返回第一种类型。因此，您提供的类型顺序非常重要。

req.accepts(types)

检查给定的类型 `types(s)` 是否可被接受，当为 true 时返回最佳匹配，否则返回 `false`。`type` 的值可以是一个或者多个 mime 类型字符串。 比如 "application/json" 扩展名为 "json"，或者数组 `["json", "html", "text/plain"]`。

```
// Accept: text/html
this.accepts('html');
// => "html"

// Accept: text/*, application/json
this.accepts('html');
// => "html"
this.accepts('text/html');
// => "text/html"
this.accepts('json', 'text');
// => "json"
this.accepts('application/json');
// => "application/json"

// Accept: text/*, application/json
this.accepts('image/png');
this.accepts('png');
// => false

// Accept: text/*;q=.5, application/json
this.accepts(['html', 'json']);
this.accepts('html', 'json');
// => "json"

// No Accept header
this.accepts('html', 'json');
// => "html"
this.accepts('json', 'html');
// => "json"
```

`this.accepts()` 可以被调用多次，或者使用 switch:

```
switch (this.accepts('json', 'html', 'text')) {
  case 'json': break;
  case 'html': break;
  case 'text': break;
  default: this.throw(406, 'json, html, or text only');
}
```

req.acceptsEncodings(encodings)

检查 `encodings` 是否可以被接受，当为 `true` 时返回最佳匹配，否则返回 `false` 。 注意：您应该在 encodings 中包含 `identity` 。

```
// Accept-Encoding: gzip
this.acceptsEncodings('gzip', 'deflate', 'identity');
// => "gzip"

this.acceptsEncodings(['gzip', 'deflate', 'identity']);
// => "gzip"
```

当没有传递参数时，返回包含所有可接受的 encodings 的数组：

```
// Accept-Encoding: gzip, deflate
this.acceptsEncodings();
// => ["gzip", "deflate", "identity"]
```

注意：如果客户端直接发送 `identity;q=0` 时，`identity` encoding（表示no encoding） 可以不被接受。虽然这是一个边界情况，您仍然应该处理这种情况。

req.acceptsCharsets(charsets)

检查 `charsets` 是否可以被接受，如果为 `true` 则返回最佳匹配， 否则返回 `false` 。

```
// Accept-Charset: utf-8, iso-8859-1;q=0.2, utf-7;q=0.5
this.acceptsCharsets('utf-8', 'utf-7');
// => "utf-8"

this.acceptsCharsets(['utf-7', 'utf-8']);
// => "utf-8"
```

当没有传递参数时， 返回包含所有可接受的 charsets 的数组：

```
// Accept-Charset: utf-8, iso-8859-1;q=0.2, utf-7;q=0.5
this.acceptsCharsets();
// => ["utf-8", "utf-7", "iso-8859-1"]
```

req.acceptsLanguages(langs)

检查 `langs` 是否可以被接受，如果为 `true` 则返回最佳匹配，否则返回 `false` 。

```
// Accept-Language: en;q=0.8, es, pt
this.acceptsLanguages('es', 'en');
// => "es"

this.acceptsLanguages(['en', 'es']);
// => "es"
```

当没有传递参数时，返回包含所有可接受的 langs 的数组：

```
// Accept-Language: en;q=0.8, es, pt
this.acceptsLanguages();
// => ["es", "pt", "en"]
```

req.idempotent

检查请求是否为幂等(idempotent)。

req.socket

返回请求的socket。

req.get(field)

返回请求 header 中对应 field 的值。

响应(Response)

Koa `Response` 对象是对 node 的 response 进一步抽象和封装，提供了日常 HTTP 服务器开发中一些有用的功能。

API

res.header

Response header 对象。

res.socket

Request socket。

res.status

获取 response status。不同于 node 在默认情况下 `res.statusCode` 为200，`res.status` 并没有赋值。

res.statusString

Response status 字符串。

res.status=

通过 数字状态码或者不区分大小写的字符串来设置response status：

- 100 "continue"
- 101 "switching protocols"
- 102 "processing"
- 200 "ok"
- 201 "created"
- 202 "accepted"
- 203 "non-authoritative information"
- 204 "no content"
- 205 "reset content"
- 206 "partial content"
- 207 "multi-status"

- 300 "multiple choices"
- 301 "moved permanently"
- 302 "moved temporarily"
- 303 "see other"
- 304 "not modified"
- 305 "use proxy"
- 307 "temporary redirect"
- 400 "bad request"
- 401 "unauthorized"
- 402 "payment required"
- 403 "forbidden"
- 404 "not found"
- 405 "method not allowed"
- 406 "not acceptable"
- 407 "proxy authentication required"
- 408 "request time-out"
- 409 "conflict"
- 410 "gone"
- 411 "length required"
- 412 "precondition failed"
- 413 "request entity too large"
- 414 "request-uri too large"
- 415 "unsupported media type"
- 416 "requested range not satisfiable"
- 417 "expectation failed"
- 418 "i'm a teapot"
- 422 "unprocessable entity"
- 423 "locked"
- 424 "failed dependency"
- 425 "unordered collection"
- 426 "upgrade required"
- 428 "precondition required"
- 429 "too many requests"
- 431 "request header fields too large"
- 500 "internal server error"
- 501 "not implemented"
- 502 "bad gateway"
- 503 "service unavailable"
- 504 "gateway time-out"
- 505 "http version not supported"

- 506 "variant also negotiates"
- 507 "insufficient storage"
- 509 "bandwidth limit exceeded"
- 510 "not extended"
- 511 "network authentication required"

注意：不用担心记不住这些字符串，如果您设置错误，会有异常抛出，并列出该状态码表来帮助您进行更正。

res.length=

通过给定值设置 response Content-Length。

res.length

如果 Content-Length 作为数值存在，或者可以通过 `res.body` 来进行计算，则返回相应数值，否则返回 `undefined` 。

res.body

获得 response body。

res.body=

设置 response body 为如下值：

- `string` written
- `Buffer` written
- `Stream` piped
- `Object` json-stringified
- `null` no content response

如果 `res.status` 没有赋值，Koa会自动设置为 `200` 或 `204` 。

String

Content-Type 默认为 text/html 或者 text/plain，两种默认 charset 均为 utf-8。 Content-Length 同时会被设置。

Buffer

Content-Type 默认为 application/octet-stream，Content-Length同时被设置。

Stream

Content-Type 默认为 application/octet-stream。

Object

Content-Type 默认为 application/json。

res.get(field)

获取 response header 中字段值，field 不区分大小写。

```
var etag = this.get('ETag');
```

res.set(field, value)

设置 response header 字段 `field` 的值为 `value` 。

```
this.set('Cache-Control', 'no-cache');
```

res.set(fields)

使用对象同时设置 response header 中多个字段的值。

```
this.set({
  'Etag': '1234',
  'Last-Modified': date
});
```

res.remove(field)

移除 response header 中字段 `filed` 。

res.type

获取 response `Content-Type` ，不包含像 "charset" 这样的参数。

```
var ct = this.type;
// => "image/png"
```

res.type=

通过 mime 类型的字符串或者文件扩展名设置 response `Content-Type`

```
this.type = 'text/plain; charset=utf-8';
this.type = 'image/png';
this.type = '.png';
this.type = 'png';
```

注意：当可以根据 `res.type` 确定一个合适的 `charset` 时，`charset` 会自动被赋值。 比如 `res.type = 'html'` 时，charset 将会默认设置为 "utf-8"。然而当完整定义为 `res.type = 'text/html'` 时，charset 不会自动设置。

res.redirect(url, [alt])

执行 [302] 重定向到对应 `url` 。

字符串 "back" 是一个特殊参数，其提供了 Referrer 支持。当没有Referrer时，使用 `alt` 或者 `/` 代替。

```
this.redirect('back');
this.redirect('back', '/index.html');
this.redirect('/login');
this.redirect('http://google.com');
```

如果想要修改默认的 [302] 状态，直接在重定向之前或者之后执行即可。如果要修改 body，需要在重定向之前执行。

```
this.status = 301;
this.redirect('/cart');
this.body = 'Redirecting to shopping cart';
```

res.attachment([filename])

设置 "attachment" 的 `Content-Disposition` ，用于给客户端发送信号来提示下载。filename 为可选参数，用于指定下载文件名。

res.headerSent

检查 response header 是否已经发送，用于在发生错误时检查客户端是否被通知。

res.lastModified

如果存在 `Last-Modified` ，则以 `Date` 的形式返回。

res.lastModified=

以 UTC 格式设置 `Last-Modified` 。您可以使用 `Date` 或 date 字符串来进行设置。

```
this.response.lastModified = new Date();
```

res.etag=

设置 包含 `"`s 的 ETag。注意没有对应的 `res.etag` 来获取其值。

```
this.response.etag = crypto.createHash('md5').update(this.body).digest('hex');
```

res.append(field, val)

在 header 的 `field` 后面 追加 `val` 。

res.vary(field)

相当于执行res.append('Vary', field)。

相关资源

Community links to discover third-party middleware for Koa, full runnable examples, thorough guides and more! If you have questions join us in IRC! 以下列出了更多第三方提供的 koa 中间件、完整实例、全面的帮助文档等。如果有问题，请加入我们的 IRC！

- GitHub repository
- Examples
- Middleware
- Wiki
- G+ Community
- Mailing list
- Guide
- FAQ
- **#koajs** on freenode